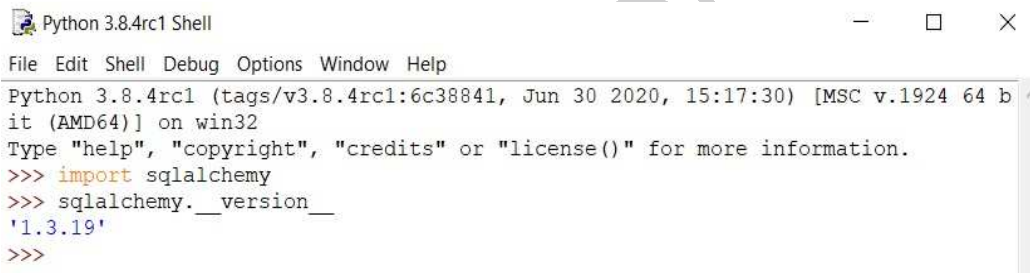


Upotreba okvira SQLAlchemy

SQLAlchemy je popularan SQL alat i ORM (object-relational mapper). To je softver otvorenog koda i višepatformski softver napisan u Pythonu i objavljen pod MIT licencom. SQLAlchemy je poznat po svom objektno-relacionom mapiranju (ORM), pomoću kojeg se klase mogu preslikati u bazu podataka, omogućavajući tako objektnom modelu i šemi baze podataka da se od početka razvijaju na čisto odvojeni način.

Kako veličina i performanse SQL baza podataka postaju sve bitnije, one se ponašaju manje kao kolekcije objekata. S druge strane, kako apstrakcija u kolekcijama predmeta počinje da važi, oni se ponašaju manje poput tabela i redova. Cilj SQLAlchemyja je da prilagodi oba ova principa. Iz tog razloga je usvojio obrazac mapiranja podataka, a ne aktivni obrazac zapisa koji koriste brojni drugi ORM-ovi.

Za korišćenje SQLAlchemyja potrebna je njegova instalacija, koja se izvršava naredbom `pip install sqlalchemy` u komandnoj liniji. Kako bismo proverili da li je instalacija uspešno izvršena, importovaćemo ovaj modul u naše Python komandno okruženje i pozvati komandu za upit verzije (slika 11.1).



```
Python 3.8.4rc1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.4rc1 (tags/v3.8.4rc1:6c38841, Jun 30 2020, 15:17:30) [MSC v.1924 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import sqlalchemy
>>> sqlalchemy.__version__
'1.3.19'
>>>
```

Slika 11.1. Provera instalacije SQLAlchemy modula

SQLAlchemy koristi dijalekatski sistem za komunikaciju sa različitim vrstama DB-API implementacija i relacionih baza podataka kao što su MS SQL, MySQL, Oracle SQL, PostgreSQL, SQLite, SyBase i Firebird. Kao što smo već napomenuli u lekciji *Manipulacija konekcijom i čuvanje konekcija*, DB-API specifikacija predstavlja standardni interfejs za baze podataka u Pythonu. Svi dijalekti zahtevaju instaliranje odgovarajućeg upravljačkog programa DB-API.

Izvorni SQLAlchemy podržava program za renderovanje SQL (klasa Engine), integraciju DB-API interfejsa, integraciju transakcija i šeme podataka. On koristi jezik izraza (expression language) koji se aplicira na šemu podataka, dok je njegov ORM više usmeren na domen. Drugim rečima, **Expression Language** je jedna od osnovnih komponenti SQL Alchemyja, koja važi za sistem predstavljanja relacionih struktura baze podataka i izraza pomoću Python konstrukcija, za razliku od **ORM**-a, koji predstavlja visok nivo i apstraktni obrazac upotrebe, koji je sam po sebi primenjena upotreba jezika izraza. Naredbe jezika izraza se u SQLAlchemy programu za renderovanje prevode u odgovarajuće SQL upite.

Konekcija

Za konektovanje modula SQLAlchemy na bazu podataka koristi se ranije pomenuta **Engine** klasa tog modula. Za uspostavljanje izvora konekcije i ponašanja baze podataka, Engine klasa spaja **bazu konekcija** i **dijalekat**.

Napomena

Baza konekcija (connection pool) je skup konekcija skladištenih u keš memoriji tako da se veze mogu ponovo koristiti kada baza podataka primi buduće zahteve za podacima. Nakon stvaranja veze, ona se stavlja u bazu i ponovo koristi, tako da nova veza ne mora biti uspostavljena svaki put. Ovaj mehanizam se koristi za poboljšanje performansi izvršavanja naredbi u bazi podataka.

Objekat klase `Engine` se instancira pozivanjem metode `create_engine()`. Ova funkcija kao argument uzima bazu podataka koja će biti kreirana. Standardna forma pozivanja ove funkcije zahteva da baza podataka bude prosleđena u URL formatu koji predstavlja podatak tipa string i sadrži neophodne informacije o dijalektu baze i detaljima konekcije. Parametar `echo` predstavlja prečicu za podešavanje logovanja, što se postiže putem Pythonovog standardnog logging modula. Kada je ovaj parametar postavljen na vrednost `yes`, to znači da će se na ekranu ispisivati rezultati svih izvršenih upita. U nastavku su dati primeri konekcija na SQLite i MySQL bazu podataka u SQLAlchemy modulu.

Primer konekcije na SQLite bazu podataka korišćenjem `create_engine()` funkcije SQLAlchemy modula:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///example.db', echo = True)
```

Primer konekcije na MySQL bazu podataka korišćenjem `create_engine()` funkcije SQLAlchemy modula:

```
from sqlalchemy import create_engine
engine = create_engine("mysql://user:pwd@localhost/example", echo = True)
```

Forma URL-a koji je prosleđen u vidu stringa predstavlja DB-API interfejs koji baza podataka koristi. Radi lakšeg razumevanja načina na koju su parametri u ovom linku definisani, u nastavku prikazujemo njegovu uopštenu strukturu:

```
dialect[+driver]://user:password@host/dbname
```

Kreiranje baze podataka za sve SUBP koje SQLAlchemy podržava moguće je videti na zvaničnom linku [SQLAlchemy dokumentacije](#). U nastavku su prikazane neke od najvažnijih funkcija klase `Engine` (tabela 11.1).

Funkcije <code>Engine</code> klase u <code>SQLAlchemy</code> modulu	
Funkcija	Opis
<code>connect()</code>	vraća objekat veze
<code>execute()</code>	izvršava SQL naredbu
<code>begin()</code>	prikazuje menadžer konteksta koji isporučuje vezu sa uspostavljenom transakcijom; nakon uspešne operacije, transakcija se izvršava, u suprotnom se vraća nazad
<code>dispose()</code>	odlaže bazu konekcija koju kreira <code>Engine</code>
<code>driver()</code>	ime drajvera dijalekta koji koristi <code>Engine</code>

Tabela 11.1. Funkcije `Engine` klase u `SQLAlchemy` modulu

Kreiranje tabela

SQL Expression Language izvršava svoje naredbe prema kolonama tabele. Objekat `Column` ovog modula predstavlja kolonu u tabeli baze podataka, koja je s druge strane predstavljena objektom `Table`. Definisane tebele i pridruženi objekti kao što su indeksi, pogledi, okidači i sl. predstavljaju kolekciju metapodataka. Otuda objekat klase `MetaData` čija je uloga skladištenje kolekcije objekata tabela, kao i opcionog povezivanja sa mašinom (`Engine`) ili konekcijom (`Connection`).

Najpre je potrebno da instanciramo objekat metapodataka. Pozivanjem `MetaData()` konstruktora kreira se nova instanca ovog objekta, čija je svrha povezivanje podataka sa šemom. Ukoliko se konstruktoru ne proslede parametri, njihova vrednost će podrazumevano biti **None**. Instancu ćemo potom sačuvati u promenljivu.

Sledeći korak je kreiranje tabele, koja u vidu parametara prima metapodatake, ime tabele koja će se kreirati i objekte kolona. Objekti kolona sadrže naziv kolone, tip podatka i ograničenje nad tim podatkom. Listu generičkih tipova podataka podržanih u `SQLAlchemy`ju možemo proveriti [ovde](#). Za korišćenje ovih tipova podataka potrebno je uvođenje njihovih klasa u program. Kreiranu tabelu potrebno je sačuvati u promenljivu.

Poslednji korak u procesu kreiranja tabele je pozivanje metode `createall()` objekta metapodataka, koja za parametar prima objekat `engine` čija je uloga kreiranje podataka i njihovo skladištenje u metapodatke. U primeru koji sledi prikazano je kako izgleda postupak kreiranja tabele `studenata` u `SQLite` bazi podataka. Pošto smo parametar `echo` funkcije kreiranja postavili na vrednost `True`, na ekranu će nam se prikazati izvršeni upit u SQL upitnom jeziku.

Primer kreiranja tabele u `SQLite` bazi podataka korišćenjem `SQLAlchemy` modula:

```
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String

engine = create_engine('sqlite:///example.db', echo = True)
meta = MetaData()

students = Table(
    'students', meta,
```

```

        Column('id', Integer, primary_key = True),
        Column('name', String),
        Column('address', String),
    )

    meta.create_all(engine)

```

```

CREATE TABLE students (
    id INTEGER NOT NULL,
    name VARCHAR,
    address VARCHAR,
    PRIMARY KEY (id)
)

```

Slika 11.2. Prikaz rezultata izvršenja SQLAlchemy naredbe u SQL formatu

Izmeniti kod tako da se u bazi example.db napravite tabelu School pomoću sqlalchemy modula koja bi trebalo da sadrži attribute: school_name(string, predstavlja primarni ključ za ovu tabelu), level_of_education(string), adress(string), phone_number(string).

Izvršenje SQL naredbi

SQL naredbe se izvršavaju pomoću odgovarajućih metoda za kreiranje upita, u odnosu na ciljani objekat tabele. U ove naredbe spadaju CRUD operacije – insert, select, update i delete. Njihovo izvršavanje u SQLAlchemy modulu pozivamo preko istoimenih metoda. Sve naredbe u SQLAlchemyju se pozivaju preko objekta tabele. U našem primeru, pozivanje naredbe za umetanje podataka izgledalo bi ovako:

```
query = student.insert()
```

Ova komanda predstavlja zamenu za klasičnu, dole navedenu, SQL naredbu, koja je u stvari njen ekvivalent, što se može potvrditi pomoću funkcije `str(ins)`, koja kao parametar prima naš upit nad bazom i prikazuje ga u formatu klasičnog SQL jezika upita:

```

'INSERT INTO students (student_id, name, address)
VALUES (:student_id, :name, :address)'

```

Za umetanje specifičnih vrednosti pozivamo metodu `values()` iz objekta koji želimo da umetnemo.

```

query = students.insert().values(name = 'Bob Johnson')
'INSERT INTO students (name) VALUES (:name)'

```

U ovom slučaju, konkretna vrednost parametra se ne prikazuje. Umesto toga, SQLAlchemy generiše vezane parametre koji su vidljivi u kompajliranoj formi izraza:

```
query.compile().params
```

```
{'name': 'Bob Johnson'}
```

Da bi se prikazani upiti izvršili, neophodno je da imamo aktivno provereni izvor DB-API veze, što postizemo komandom `engine.connect()`. Pomoću uspostavljene konekcije, nakon umetanja podataka u bazu, pozvamo metodu izvršenja `execute()`, koja će izvršiti trajne promene na bazi podataka:

```
conn = engine.connect()
query = students.insert().values(
    name = 'Bob Johnson', address = 'Green Avenue 2331')
result = conn.execute(query)

INSERT INTO students (name, lastname) VALUES (?, ?)
('Bob Johnson', 'Green Avenue 2331')
COMMIT
```

U nastavku je dat primer kako u svoju tabelu studenata dodajemo jednog novog studenta.

Primer umetanja podataka u tabelu SQLite baze korišćenjem SQLAlchemy modula:

```
query = students.insert()
query = students.insert().values(name = 'Bob Johnson',
                                address = 'Green avenue 2331')

conn = engine.connect()
result = conn.execute(query)
```

Videli smo kako izgleda primer sa umetanjem jednog reda u tabelu baze podataka. Ako bismo želeli da umetnemo više zapisa odjednom, podatke bismo prosledili u vidu liste rečnika podataka, što bi izgledalo kao u sledećem primeru.

Primer umetanja više redova podataka u tabelu SQLite baze korišćenjem SQLAlchemy modula:

```
conn.execute(students.insert(), [
    {'name': 'Jessica Sims', 'address' : '5th Avenue 27'},
    {'name': 'Adam Benson', 'address' : 'Wallstreet 147'},
    {'name': 'Melissa Jenkins', 'address' : 'Boulevard 654'},
    {'name': 'Kevin Robertson', 'address' : '6th Avenue 103'},
])
```

Za prikazivanje podataka koristimo `select()` metodu za kreiranje upita:

```
query = students.select()
'SELECT students.id, students.name, students.address FROM students'
```

Za trajno izvršenje promena nad bazom, kao i kod `insert` metode, koristimo funkciju `execute()`.

```
result = conn.execute(query)
SELECT students.id, students.name, students.address
FROM students
```

Ako obratimo pažnju, primetićemo da je promenljiva `result` koju smo definisali sada zapravo dobila ulogu kursora. Stoga je preko ove promenljive moguće pozivanje `fetchone()` funkcije koja vraća rezultat sledećeg reda upita, dok je pregled svih redova moguć korišćenjem `for` petlje. U nastavku je prikazana primena upita za prikazivanje podataka u SQLite bazi.

Primer prikazivanja podataka iz tabele SQLite baze korišćenjem SQLAlchemy modula:

```
query = students.select()
conn = engine.connect()
result = conn.execute(query)

for row in result:
    print (row)
```

Prikazivanje podataka iz tabele može da se vrši i na osnovu zadatih uslova za prikaz, koji se definišu pomoću `where()` funkcije kojoj se ti uslovi prosleđuju. U nastavku ćemo prikazati kako bismo od programa tražili da vrati samo studente čije je ime *Melissa Jenkins*. Atribut `c` u parametru `students.c.name` predstavlja alijas za kolonu.

Primer prikazivanja podataka + iz tabele SQLite baze sa zadatim uslovom korišćenjem SQLAlchemy modula:

```
conn = engine.connect()
query = students.select().where(students.c.name=='Melissa Jenkins')
result = conn.execute(query)

for row in result:
    print (row)
```

Rezultat: (4, 'Melissa Jenkins', 'Boulevard 654')

Izmeniti kod tako da se u bazi `example.db` napravite tabelu `Clothes` pomoću `sqlalchemy` modula koja bi trebalo da sadrži attribute: `product_id`(ceo broj, predstavlja primarni ključ za ovu tabelu), `type`(string), `color`(string), `price`(realan broj). Uneti podatke za 3 proizvoda.

Za ažuriranje podataka u bazi koristimo sledeću sintaksu:

```
table.update().where(conditions).values(SET expressions)
```

Metoda upita `update()` se poziva iz tabele nad kojom želimo da izvršimo ažuriranje. U okviru metode ažuriranja postavljamo uslov za promenu podataka u vidu parametra funkcije

`where()` i nove vrednosti podataka u vidu parametara funkcije `values()`. Ovako bi izgledao primer ažuriranja adrese jednog studenta.

Primer ažuriranja podataka iz tabele SQLite baze korišćenjem SQLAlchemy modula:

```
conn = engine.connect()

query = students.update().where(
    students.c.address == 'Wallstreet 147').values(address = 'Floorstreet
5')

conn.execute(query)
```

```
'UPDATE students SET address = :address
WHERE students.address = :address_1'
```

Još jedna od metoda upita koje SQLAlchemy podržava je brisanje podataka, odnosno `delete()`, koja se izvršava na isti način kao i ostale, pozivanjem iz objekta tabele. Ovako izgledaju pozivanje metode brisanja u SQLAlchemyju i njen ekvivalent u SQL jeziku upita:

```
stmt = students.delete()
'DELETE FROM students'
```

Ova metoda upita takođe može da prihvati i uslov pod kojim će se operacija brisanja izvršiti, i to čini prihvatanjem uslova u vidu parametra metode `where()`.

Primer brisanja podataka iz tabele SQLite baze korišćenjem SQLAlchemy modula:

```
conn = engine.connect()

query = students.delete().where(students.c.name == 'Jessica Sims')

conn.execute(query)
```

```
'DELETE FROM students WHERE students.name = :name_1'
```

U nastavku je sumiran prikaz svih naredbi koje su nad našom bazom izvršene. Na osnovu priloženog rezultata zaključujemo da su promene na bazi uspešno aplicirane.

Rezultat: [(1, 'Bob Johnson', 'Green avenue 2331'), (3, 'Adam Benson', 'Floorstreet 5'), (4, 'Melissa Jenkins', 'Boulevard 654'), (5, 'Kevin Robertson', '6th Avenue 103')]

Sumiran prikaz korišćenja CRUD operacija nad SQLite bazom podataka u SQLAlchemy modulu:

```
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String

engine = create_engine('sqlite:///example.db', echo = True)
meta = MetaData()

students = Table(
    'students', meta,
    Column('id', Integer, primary_key = True),
    Column('name', String),
    Column('address', String),
)

meta.create_all(engine)

query1 = students.insert()
query1 = students.insert().values(name = 'Bob Johnson',
                                   address = 'Green avenue 2331')

conn = engine.connect()

conn.execute(query1)

conn.execute(students.insert(), [
    {'name': 'Jessica Sims', 'address' : '5th Avenue 27'},
    {'name': 'Adam Benson', 'address' : 'Wallstreet 147'},
    {'name': 'Melissa Jenkins', 'address' : 'Boulevard 654'},
    {'name': 'Kevin Robertson', 'address' : '6th Avenue 103'},
])

query2 = students.update().where(
    students.c.address == 'Wallstreet 147').values(address = 'Floorstreet 5')

conn.execute(query2)

query3 = students.delete().where(students.c.name == 'Jessica Sims')

conn.execute(query3)

s = students.select()
data = conn.execute(s).fetchall()
print(data)
```

Izmeniti kod tako da se u bazi example.db napravite tabelu Clothes pomoću sqlalchemy modula koja bi trebalo da sadrži atribut: product_id(ceo broj, predstavlja primarni ključ za ovu tabelu), type(string), color(string), price(realan broj). Uneti podatke za haljinu, pantalone i kaput. Potrebno je da promenite cenu za proizvod haljine a da izbrišete proizvod pantalona.

Pitanje

Za povezivanje na bazu podataka u SQLAlchemy modulu koristi se funkcija:

- `connect()`
- `execute()`
- **`create_engine()`**
- `conn.execute()`

Objašnjenje:

Za konekciju na bazu podataka u SQLAlchemy modulu koristi se funkcija `create_engine()`.

SQLAlchemy ORM

Glavni cilj SQLAlchemy ORM-a je da olakša povezivanje korisnički definisanih Python klasa sa tabelama baze podataka, a objekata tih klasa sa redovima u odgovarajućim tabelama. SQLAlchemy ORM je napravljen na vrhu jezika SQL izraza. On je zapravo visok i apstrahovan obrazac upotrebe, odnosno, primenjena upotreba jezika izraza. Iako se uspešna aplikacija može konstruisati koristeći isključivo object-relational mapper, ponekad aplikacija izrađena sa ORM-om može direktno koristiti jezik izraza tamo gde su potrebne određene interakcije sa bazom podataka.

Kod ORM-a, proces konfiguracije započinje opisivanjem tabela baze podataka, a zatim definisanjem klasa koje će se preslikati u te tabele. U SQLAlchemyju, ova dva zadatka se izvode istovremeno. To se postiže korišćenjem sistema deklaracije, u kom stvorene klase uključuju direktive za opis stvarne tabele baze podataka u koju su mapirane. Klasa koja čuva katalog klasa i mapiranih tabela u sistemu deklaracije naziva se deklarativna ili bazna klasa. Funkcija `declarative_base()` koristi se za njeno kreiranje. Ova funkcija je definisana u `sqlalchemy.ext.declarative` modulu. Jednom kada se bazna klasa deklariše, bilo koji broj mapiranih klasa može se definisati iz nje.

Kao i do sada, da bismo pristupili radu sa bazom podataka, potrebna nam je konekcija na nju. U naš program ćemo pored deklarativnog modula uvesti i module tipova podataka, kao i engine modul. Dalje u programu ćemo definisati klasu `Students`, koja sadrži tabelu u koju se mapira i imena i tipove podataka kolona u njoj. Zatim ćemo korišćenjem `create_all()` funkcije iz metapodataka bazne klase narediti kreiranje definisane tabele.

Primer kreiranja bazne klase i njenog mapiranja u bazu podataka korišćenjem SQLAlchemy ORM-a:

```
from sqlalchemy import Column, Integer, String
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///example_orl.db', echo = True)
Base = declarative_base()

class Students(Base):
    __tablename__ = 'students'
```

```

id = Column(Integer, primary_key = True)
name = Column(String)
address = Column(String)

Base.metadata.create_all(engine)

```

Da bismo mogli da komuniciramo sa bazom podataka, moramo da dobijemo njen upravljač. Ovu funkcionalnost nam omogućava objekat sesije. Klasa `Session` se definiše pomoću `sessionmaker()` konfigurabilne metode fabričke sesije koja je vezana za prethodno napravljeni objekat `engine`. Ovako kreirana sesija potom koristi prazni konstruktor.

Primer kreiranja sesije korišćenjem SQLAlchemy ORM-a:

```

from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind = engine)
session = Session()

```

U nastavku je dat pregled nekih od najčešće upotrebljavanih funkcija sesije.

Funkcije <code>Session</code> klase u SQLAlchemy ORM-u	
Funkcija	Opis
<code>begin()</code>	započinje transakciju u ovoj sesiji
<code>add()</code>	postavlja objekat u sesiju
<code>add_all()</code>	dodaje kolekciju objekata u sesiju
<code>commit()</code>	izvršava sve transakcije uključujući i one koje su u procesu
<code>delete()</code>	briše transakciju
<code>execute()</code>	izvršava SQL izraz
<code>expire()</code>	označava atribut instance kao nevažeći
<code>flush()</code>	aplicira sve promene na bazu

Tabela 11.2. Funkcije `Session` klase u SQLAlchemy modulu

Pošto smo kreirali sesiju i deklarirali klasu studenata koja je preslikana u tabelu, sledeći korak je dodavanje objekata ove klase metodom `add()` sesijskog objekta. U okviru ove metode potrebno je u vidu parametara dodeliti vrednosti kolonama tabele, a zatim tako definisan objekat smestiti u promenljivu. Promenljivu objekta ćemo zatim proslediti u vidu parametra funkciji `commit()`, koja će trajno upisati ovaj objekat u našu bazu.

Primer dodavanja reda u tabelu korišćenjem SQLAlchemy ORM-a:

```

s1 = Students(name = 'Bob Johnson', address = 'Green Avenue 2331')
session.add(s1)
session.commit()

```

SQLAlchemy modul omogućava dodavanje više redova u tabelu istovremeno korišćenjem funkcije `add_all()`, čiju je primenu moguće videti u sledećem primeru.

Primer dodavanja više redova u tabelu korišćenjem SQLAlchemy ORM-a:

```
session.add_all([
    Students(name='Jessica Sims', address='5th Avenue 27'),
    Students(name='Adam Benson', address='Wallstreet 147'),
    Students(name='Melissa Jenkins', address='Boulevard 654'),
    Students(name='Kevin Robertson', address='6th Avenue 103')]
)

session.commit()
```

Sve SELECT upite koje generiše SQLAlchemy ORL konstruiše objekat upita Query. Svakim pozivom upita generiše se novi objekat koji predstavlja kopiju prethodnog objekta modifikovanog po osnovu zadatog kriterijuma upita.

Objekti upita su inicijalno generisani u metodi `query()`, koju je moguće pozvati iz objekta sesije. Uopšteni prikaz korišćenja objekta upita je:

```
q = session.query(mapped class)
```

a njemu ekvivalentan izraz:

```
q = Query(mappedClass, session)
```

Za vraćanje liste objekata iz tabele koristimo funkciju `all()`. U nastavku ćemo prikazati kako pomoću ove funkcije vršimo ispis naše liste studenata.

Primer prikazivanja redova tabele korišćenjem SQLAlchemy ORM-a:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind = engine)
session = Session()
result = session.query(Students).all()

for row in result:
    print ("Name: ", row.name, "Address:", row.address)
```

```
FROM students
2020-09-03 19:22:35,506 INFO sqlalchemy.engine.base.Engine ()
Name: Bob Johnson Address: Green Avenue 2331
Name: Jessica Sims Address: 5th Avenue 27
Name: Adam Benson Address: Wallstreet 147
Name: Melissa Jenkins Address: Boulevard 654
Name: Kevin Robertson Address: 6th Avenue 103
```

Slika 11.3. Rezultati prikazivanja redova tabele u okviru sesije u SQLAlchemy modulu

Funkcije Query klase u SQLAlchemy ORM-u	
Funkcija	Opis
<code>add_columns()</code>	dodaje jednu ili više kolona na listu rezultata koje treba vratiti
<code>add_entity()</code>	dodaje mapirani entitet na listu rezultata koje treba vratiti
<code>count()</code>	vraća broj redova upita
<code>delete()</code>	briše redove dobijene upitom
<code>distinct()</code>	vraća samo različite vrednosti iz upita
<code>filter()</code>	primenjuje dati kriterijum filtriranja na kopiju upita
<code>first()</code>	vraća prvi red rezultata upita; ako upit ne vraća nijednu vrednost, ispisaće se Null
<code>get()</code>	vraća instancu zasnovanu na datom identifikatoru primarnog ključa pružajući direktan pristup mapi identiteta sesije koja ga poseduje
<code>group_by()</code>	aplicira kriterijume grupisanja podataka i vraća nove podatke
<code>join()</code>	spaja tabele na osnovu zadatog upita
<code>one()</code>	vraća tačno jedan rezultat ili javlja grešku
<code>order_by()</code>	aplicira kriterijum sortiranja
<code>update()</code>	ažurira upite koji odgovaraju zadatom kriterijumu

Tabela 11.3. Funkcije Query klase u SQLAlchemy modulu

Sve funkcionalnosti koje smo primenjivali u vezi sa SQLAlchemy ORL-om sumiraćemo u primeru ispod, prema redosledu njihovog izvršavanja.

Sumiran prikaz korišćenja osnovnih SQLAlchemy ORL funkcionalnosti:

```

from sqlalchemy import Column, Integer, String
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///example_orl.db', echo = True)
Base = declarative_base()

class Students(Base):
    __tablename__ = 'students'

    id = Column(Integer, primary_key = True)
    name = Column(String)
    address = Column(String)

Base.metadata.create_all(engine)
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind = engine)
session = Session()

s1 = Students(name = 'Bob Johnson', address = 'Green Avenue 2331')
session.add(s1)

session.add_all([
    Students(name='Jessica Sims', address='5th Avenue 27'),
    Students(name='Adam Benson', address='Wallstreet 147'),
    Students(name='Melissa Jenkins', address='Boulevard 654'),
    Students(name='Kevin Robertson', address='6th Avenue 103')])

```

```

    )

    session.commit()
    result = session.query(Students).all()

    for row in result:
        print ("Name: ",row.name, "Address:",row.address)

```

Zadatak za vežbu

Korišćenjem SQLAlchemy okvira i SQLite baze podataka, kreirati tabelu Playlist koja će sadržati sledeće kolone:

- id – primarni ključ;
- author – text;
- song – text.

U kreiranu tabelu potrebno je omogućiti unošenje informacija u postojeće kolone iz komandne linije korišćenjem funkcije `input()`.

Nakon unošenja pesme, neophodno je prikazati sve rezultate.

Rešenje:

```

from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String

engine = create_engine('sqlite:///Playlist.db', echo = False)
meta = MetaData()

playlist = Table(
    'playlist', meta,
    Column('id', Integer, primary_key = True),
    Column('author', String),
    Column('song', String),
)
meta.create_all(engine)

song_id = input("Enter song id: ")
song_author = input("Enter author: ")
song_name = input("Enter song: ")

query = playlist.insert().values(id = song_id, author= song_author, song
= song_name)

query2 = playlist.select()

conn = engine.connect()
r1 = conn.execute(query)
result = conn.execute(query2)

for row in result:
    print("Author: {} | Song: {}".format(row[1], row[2]))

```

Objašnjenje:

Prvi korak jeste uključivanje potrebnih SQLAlchemy modula. Nakon toga, potrebno je da kreiramo tabelu koristeći `Table` konstruktor i `create_all` metodu `MetaData` objekta. Zatim, koristeći `input()` funkciju, možemo od korisnika zatražiti unos podataka i te podatke sačuvati. Kada su svi podaci popunjeni, koristeći `insert` metodu promenljive `playlist`, koja sadrži našu novokreiranu tabelu, kreiramo `insert` upit i čuvamo unete podatke o pesmi. Pored `insert` metode, koristimo i `select` metodu za kreiranje upita za prikaz svih podataka. Na samom kraju, potrebno je izvršiti pomenute upite koristeći `execute` metodu konekcijskog objekta, a zatim petljom proći kroz listu rezultata drugog upita i prikazati sve pesme.

Rezime

- SQLAlchemy – popularni SQL alat poznat po svom objektno-relacionom mapperu (ORM), pomoću kojeg se klase mogu preslikati u bazu podataka.
- SQLAlchemy koristi dijalekatski sistem za komunikaciju sa različitim vrstama DB-API implementacija i relacionih baza podataka kao što su MS SQL, MySQL, Oracle SQL, PostgreSQL, SQLite, SyBase i Firebird.
- Expression Language je jezik izraza koji se koristi kao sistem predstavljanja relacionih struktura baze podataka i izraza pomoću Python konstrukcija.
- ORM je apstraktni obrazac upotrebe jezika izraza.
- `Engine` je klasa za uspostavljanje izvora konekcije i ponašanja baze podataka. Spaja bazu konekcija i dijalekat. Služi za prevođenje koda u jezik baze podataka SQL.
- Za konekciju na bazu podataka u SQLAlchemyju se koristi metoda `create_engine()`, koja za parametar prima URL konekcije u kom je definisan dijalekat.
- `MetaData` je klasa koja povezuje podatke sa šemom; njena svrha je skladištenje tabela.
- `Table` je klasa koja služi za kreiranje tabela. Kao parametre uzima naziv tabele, metapodatke i objekte kolona.
- `Column` je klasa koja služi za kreiranje kolona. Kao parametre uzima naziv kolone, tip podatka i ograničenja.
- Za kreiranje podataka i njihovo skladištenje u metapodatke pozivamo obaveznu funkciju `meta.create_all(engine)`
- Za konekciju na bazu podataka koristimo funkciju `engine.connect()`, koju smeštamo u promenljivu `conn`.
- Za izvršavanje upita u bazama podataka koristimo funkciju `conn.execute()`, koja za parametar prima upit. Rezultat ovog izvršenja ima iste funkcionalnost kao i kursor i poželjno ga je sačuvati u promenljivoj `result`.
- SQLAlchemy podržava četiri osnovne CRUD operacije (insert, select, update i delete).
- Pozivanje CRUD operacija u SQLAlchemy modulu vrši se preko objekta tabele. Sintaksa koja se koristi za pozivanje CRUD operacija je sledeća:
 - umetanje – `table_name.insert().values();`
 - prikazivanje – `table_name.select().where();`
 - ažuriranje – `table_name.update().where().values();`
 - brisanje – `table_name.delete().where();`

- Bazna klasa je klasa koja čuva katalog klasa i mapiranih tabela u sistemu deklaracije. Kreira se pozivanjem `declarative_base()` i prosleđuje se definisanim klasama koje želimo da mapiramo.
- Kreiranje metapodataka u baznoj klasi vrši se kroz `Base.metadata.create_all(engine)`.
- `Session` je klasa sesije koja se definiše pomoću konfigurabilne metode fabričke sesije koja je vezana za prethodno napravljeni objekat `engine`. `Session = sessionmaker(bind = engine)`. Sesija se započinje pozivanjem konstruktora `session = Session()`.
- Za umetanje novog objekta u sesiju koristimo metodu `session.add()`, kojoj prosleđujemo objekat, a zatim pomoću `session.commit()` izvršavamo trajne promene na bazi. Funkcija `add_all()` dodaje sve objekte iz liste.
- Za primenu upita nad bazom koristimo `Query(mappedClass, session)` ili ekvivalentan poziv metode `session.query(mapped class)`. Dodavanjem funkcije `.all()` na ovu tvrdnju preuzimaju se svi redovi zadate klase.

